DAM
Desarrollo de Aplicaciones Multiplataforma
2º Curso

AD
Acceso a Datos

UD 7
Bases de Datos noSQL

IES BALMIS
Dpto Informática
Curso 2019-2020
Versión 2 (11/2019)

UD7 – Bases de Datos noSQL

ÍNDICE

6. MongoDB

MongoDB es una base de datos distribuida, basada en documentos y de uso general que ha sido diseñada para desarrolladores de aplicaciones modernas y para la era de la nube. Ninguna otra ofrece un nivel de productividad de uso tan alto.

MongoDB es una base de datos documental, lo que significa que almacena datos en forma de documentos tipo JSON, que es una forma natural de concebir los datos documentales frente al tradicional modelo de filas y columnas, ya que es mucho más expresiva y potente.

El lenguaje de consulta es rico y expresivo permitiendo filtrar y ordenar por cualquier campo, independientemente de cómo esté incrustado en un documento. Admite además agregaciones (agrupaciones) y otros casos de uso modernos, como búsqueda de gráficos o texto, y búsqueda basada en información geoespacial.

Las propias consultas son también JSON, por lo que se programan fácilmente. Olvídese de concatenar cadenas para generar consultas SQL de forma dinámica.

Características

SGBD	**MongoDB**
Modelo de Datos	**Documental**
Tipo de Datos	**JSON**
Acceso	**Cliente/Servidor**
Puerto	**27017**
Client Command-Line	**mongo**
Client GUI Escritorio (Interfaz Gráfica de Usuario)	**MongoDB Compass y Robo 3T**
Client GUI Web (Interfaz Gráfica de Usuario)	**MongoAdmin http://mongodb-tools.com/tool/mongoadmin/**

6.1 Instalación

La instalación de MongoDB es bastante sencilla, los manuales facilitados en su propia web hacen que esta tarea no sea complicada.

Desde la sección de descargas nos bajaremos la versión estable actual para Community Server en la versión de nuestro sistema operativo. Veréis que disponemos de versión con y sin SSL de 64 bits. Para esta documentación se ha utilizado la versión con SSL.

Instalación para Linux
https://docs.mongodb.com/manual/administration/install-on-linux/
Instalación para Windows
https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/
Instalación para OS X
https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/

Terminados los pasos de instalación tendremos:

- el servidor de la BD se habrá instalado en el sistema con el nombre **MongoDB**
- podemos ejecutar "**services.msc**" para Iniciar, Detener o Reiniciar el servicio
- también podremos utilizar el comando **net start "MongoDB Server"** y **net stop "MongoDB Server"**
- el programa que ejecuta el servicio es mongod
- el puerto de escucha por defecto es 27017
- se puede comprobar que está escuchando con **netstat -a -p TCP** en Windows y con **netstat -at** en Linux.

Abre un navegador y utiliza la siguiente URL http://localhost:27017/.

Activar conexiones remotas

Por defecto, comprobar con el coman do netstat que escucha sólo en "127.0.0.1". Si vas a acceder desde otro ordenador, necesitarás habilitar las conexiones remotas a tu servidor.

En Linux deberás el archivo de configuración es **/etc/mongod.conf**, y en Windows es **C:\Program Files\MongoDB\Server\4.2\bin\mongod.cfg**. Deberás buscar la línea **bindIp: 127.0.0.1** y cambiarla por **bindIp: 0.0.0.0** , ya que esto le indica que escuchará por cualquier IP.

Documentación sobre MongoDB
https://docs.mongodb.com/

6.2 Estructura y tipos de datos

La estructura de las BD noSQL es diferente a las BD Relacionales como MySQL. De todas formas, para comprender mejor su estructura, podríamos hacer las siguientes comparaciones:

MySQL	MongoDB
DataBase / Base de Datos	**DataBase** / Base de Datos
Table / Tabla	**Collection** / Colección
Record / Registro **Row** / Fila	**Document** / Documento
Column / Columna **Field** / Campo **Attribute** / Atributo	**Field** / Campo
Primary Key / Clave Primaria	_id
Relationship / Relación **Joins** / Uniones	**Embebbed documents** / Documentos embebidos **Linking /** Enlaces

MongoDB ofrece una gran variedad de tipos de datos para almacenar los valores de sus documentos. Veamos a continuación de cuales disponemos.

null

Se puede utilizar para representar tanto un valor nulo como un campo que no existe.

```
{ "x" : null }
```

boolean

Tipo booleano que permite los valores true y false.

```
{ "x" : true }
{ "y" : false }
```

number

Por defecto MongoDB utiliza números el coma flotante (Float). De esta forma los siguientes valores del campo x, son float.

```
{ "x" : 3.14 }
{ "x" : 3 }
```

Para integers o longs se tiene que indicar con sus clases propias.

```
{ "x" : NumberInt(“3”) }
{ "x" : NumberLong(“3”) }
```

string

Cualquier cadena de caracteres válidos en UTF-8 se puede representar con el tipo string.

```
{ "x" : "hola mundo" }
```

date

Las fechas se guardan en milisegundos desde la época. El time zone no se guarda.

```
{ "x" : ISODate("2014-05-25T09:09:17.027Z") }
```

Debido a que MongoDB utiliza el tipo Date de JavaScript, al crear un nuevo valor Date, es recomendable utilizar siempre new Date(), nunca Date(…), ya que esto no devolverá un objeto Date.

array

Listas de valores que se representan como arrays en MongoDB.

```
{ "x" : [ 3, 5, 9] }
```

Los arrays son listas de valores intercambiables que podremos utilizar tanto para operaciones ordenadas (colas, listas) como no ordenadas (conjuntos).
Cada valor del array puede ser del tipo que sea, incluso documentos embebidos u otros arrays.

```
{ "cosas" : [ "mesa", 3.5, true ] }
```

Una de las cosas buenas que tiene MongoDB, es que sabe como utilizar los arrays y sabe como hacer operaciones directamente sobre ellos, como buscar valores o crear índices de búsqueda, o actualizaciones de uno o varios elementos.

documentos embebidos

Los documentos pueden contener otros documentos embebidos como valor de cualquier campo.

```
{ "x" : { "a":2, "b":90} }
```

Los documentos embebidos (embedded document) se pueden utilizar para organizar la información de una manera más natural y no tan plana. Fíjate en el siguiente ejemplo.

```
{
  "nombre": "Jose Márquez",
  "direccion": {
    "calle":"Colón",
    "numero":1,
    "poblacion": "Alicante"
  }
}
```

object id

El tipo ObjectId es el tipo por defecto para el campo _id, y está diseñado para ser sencillo de generar valores únicos de forma global.

```
{ "_id" : ObjectId("4281c33ad3713f815fea36de") }
```

Esta clase está diseñada para ser ligera y fácil de generar en entornos de múltiples máquinas, entornos distribuidos, como es MongoDB. También es más sencillo de manejar que una clave autonumérica.

Se utilizarán 12 bytes para almacenamiento distribuidos de la siguiente manera.

0	1	2	3	4	5	6	7	8	9	10	11
Timestamp				**Machine**			**PID**		**Increment**		

- **Timestamp**: contienen información de fecha.
- **Machine**: viene determinado por el nombre de la máquina, generalmente un hash.
- **PID**: un identificador de proceso generado en el mismo segundo que el timestamp.
- **Increment**: es un autoincremental que permite mantener la unicidad dentro del mismo segundo.

6.3 La consola de MongoDB

Si ya tenemos funcionando nuestra base de datos MongoDB, para utilizar la consola abriremos otro terminal y en el ejecutaremos el comando mongo, es importante haber lanzado previamente el servicio. Veamos algunos comandos básicos para familiarizarnos con la consola y con este tipo de base de datos.

Documentación de The mongo Shell
https://docs.mongodb.com/manual/mongo/

Tutorial de MongoDB
https://www.tutorialspoint.com/mongodb

Consola MongoDB en línea
https://www.tutorialspoint.com/mongodb_terminal_online.php

Listar las bases de datos

```
show dbs
```

Crear una base de datos

En MongoDB no se crean las bases de datos, no existe un método como createDataBase, se crearán las bases de datos a medida que vayamos insertando la información.

No hay que preocuparse de crear una base de datos, simplemente crearemos la colección y los documentos, para eso utilizaremos las operaciones CRUD básicas más adelante.

Mostrar la base de datos actual

```
db
```

Seleccionar una base de datos

```
use nombreBD
```

A continuación se verán las operaciones CRUD principales.

Documentación MongoDB CRUD Operations (Create Read Update Delete)
https://docs.mongodb.com/manual/crud/

Crear una base de datos

Como ya se ha comentado, en MongoDB no se crean las bases de datos como tal, para ello, lo que haremos será utilizar el comando use y el nombre de la base de datos, aunque esta no exista.

```
use nombreBD
```

Insertar/guardar información en la base de datos

Con el siguiente comando, siendo la primera vez que se ejecuta, además de insertar el registro se crea la colección.

```
db.nombreColeccion.save({..})
db.nombreColeccion.insert({..})
```

La siguiente imagen muestra una inserción múltiple, en este caso, al existir ya la colección esta no vuelve a crearse.

```
db.nombreColeccion.save([{..},{..},{..}])
db.nombreColeccion.insert([{..},{..},{..}])
```

Mostrar las colecciones de la BD seleccionada

Para ver las colecciones disponibles deberemos ejecutar el siguiente comando.

```
show collections
```

Leer/mostrar una colección

Un listado completo de la colección.

```
db.nombreColeccion.find()
```

Eliminar una base de datos

Teniendo la base de datos seleccionada (con el comando use) ejecutaremos el siguiente comando.

```
db.dropDatabase()
```

Ayuda

Para entrar en la consola, disponemos de varios parámetros de configuración. Para mostrar la ayuda:

```
C:\> mongo --help
```

En la consola de mongo podemos mostrar la ayuda de comandos para una base de datos y para una colección:

```
use nombreBD
db.help()

db.nombreColeccion.help()
```

Salir y detener el servidor

Para salir, desde el terminal de MongoDB ejecutaremos exit.

```
exit
```

Por último, para detener el servidor correctamente, desde el terminal de MongoDB ejecutaremos los tres siguientes comandos.

```
use admin
db.shutdownServer()
exit
```

Ejecución de varias instrucciones desde script

Podemos escribir varias instrucciones en un archivo **miscript.js** y cargarlo con el comando load:

```
use mibasededatos
load("miscript.js")
exit
```

Veamos algunos ejemplos comparando con SQL:

MySQL	MongoDB
`CREATE DATABASE tienda;`	`use tienda`
`USE tienda;`	`use tienda`

`CREATE TABLE productos (`
`  id INT(11) PRIMARY KEY,`
`  descripcion VARCHAR(30),`
`  precio DECIMAL(10,2)`
`);`
	`db.createCollection("productos")`

`INSERT INTO`
`  productos (id, descripcion, precio)`
`VALUES`
`  (100245, 'CÁMARA FOTOS', 139.45),`
`  (200317, 'PENDRIVE 128GB', 26.35);`
	`db.productos.insert([`
	` {`
	`  "id": 100245,`
	`  "descripcion":"CÁMARA FOTOS",`
	`  "precio": 139.45`
	` },`
	` {`
	`  "id": 200317,`
	`  "descripcion":"PENDRIVE 128GB",`
	`  "precio":  26.35`
	`  }`
	`])`

`SELECT * FROM productos;`	`db.productos.find().pretty()`
`SELECT COUNT(*) FROM productos;`	`db.productos.count()`
`DROP TABLE productos;`	`db.productos.drop()`
`DROP DATABASE tienda;`	`use tienda` `db.dropDatabase()`

Consultas sobre una colección

Obtener un listado especificando un criterio. Por ejemplo para un campo:

```
db.nombreColeccion.find({campo:valor})
```

Obtener un listado utilizando expresiones regulares ($regex). Por ejemplo para un campo:

```
db.nombreColeccion.find({campo:expresionRegular})
```

Expresiones regulares para MongoDB
https://docs.mongodb.com/manual/reference/operator/query/regex/

Como comparativa con LIKE de SQL:

cadena%	**/^cadena/**
%cadena%	**/cadena/**
%cadena	**/cadena$/**

Veamos algunos ejemplos:

Mostrar los registros cuyos nombres que comiencen por 'P':

```
db.nombreColeccion.find({nombre:/^P/})
```

Mostrar los registros cuyos nombres que contengan la 'a':

```
db.nombreColeccion.find({nombre:/a/})
```

Si queremos obtener un resultado más legible, basta con añadir a find() la consulta .pretty(), de esta forma la información aparecerá más ordenada.

```
db.nombreColeccion.find().pretty()
```

Consultas con filtros

Operadores de comparación que podemos utilizar en MongoBD comparado con SQL

MDB	SQL	Descripción
eq	**=**	valores que sean iguales al valor especificado.
gt	>	valores que sean mayores que el valor especificado.
gte	>=	valores que sean iguales o mayores que el valor especificado.
lt	<	valores que sean menores que el valor especificado.
lte	<=	valores que sean iguales o menores que el valor especificado.
ne	<>	valores que sean no sean iguales que el valor especificado.
in	**in (array)**	valores que sean iguales a cualquiera de los especificados en un array.
nin	**not in (array)**	valores que no sean iguales a cualquiera de los especificados en un array.

Operadores lógicos

and	une los filtros mediante **AND** lógico, los documentos deben coincidir con las condiciones de ambos filtros.
or	une los filtros mediante **OR** lógico, los documentos deben coincidir con las condiciones de alguno de los filtros
not	invierte la expresión de la consulta, obteniendo los documentos que **NO** coincidan con el filtro.
nor	une los filtros con un NOR lógico, obtiene los documentos que no coinciden con todos los filtros.

Operadores agrupación (**$group**) respecto a un campo

min	El mínimo del grupo
max	El máximo del grupo
sum	La suma del grupo
avg	La media o valor medio del grupo

Documentación
https://docs.mongodb.com/manual/reference/operator/ https://docs.mongodb.com/manual/reference/operator/query/ https://docs.mongodb.com/manual/reference/operator/aggregation/

Veamos algunos ejemplos comparando con SQL. Creamos una colección denominada "**empleados**" en una base de datos llamada "**empresa**":

```
use empresa
db.empleados.insert(
   [
       {
         "codigo": 1,
         "nombre": "Claresta",
         "dep": "Ventas",
         "email": "cwiles0@ning.com",
         "edad": 38
       }, {
         "codigo": 2,
         "nombre": "Malinde",
         "dep": "Ventas",
         "email": "mabrahamowitcz1@paypal.org",
         "edad": 30
       }, {
         "codigo": 3,
         "nombre": "Lauri",
         "dep": "Compras",
         "email": "lharwood2@bluehost.com",
         "edad": 47
       }, {
         "codigo": 4,
         "nombre": "Richie",
         "dep": "Facturación",
         "email": "roxlee3@chronoengine.org",
         "edad": 48
       }, {
         "codigo": 5,
         "nombre": "Monti",
         "dep": "Compras",
         "email": "mcraigheid4@bandcamp.com",
         "edad": 31
       }, {
         "codigo": 6,
         "nombre": "Rosabel",
         "dep": "Ventas",
         "email": "rverrills5@java.com",
         "edad": 26
       }, {
         "codigo": 7,
         "nombre": "Caresa",
         "dep": "Ventas",
         "email": "cterrans6@bravesites.com",
         "edad": 53
       }, {
         "codigo": 8,
         "nombre": "Brendis",
         "dep": "Facturación",
         "email": "bburchfield7@cnbc.com",
         "edad": 37
       }, {
         "codigo": 9,
         "nombre": "Almeta",
         "dep": "Compras",
         "email": "alow8@dyndns.org",
         "edad": 52
       }, {
         "codigo": 10,
         "nombre": "Berkeley",
         "dep": "Ventas",
         "email": "bdacey9@wiley.com",
         "edad": 33
       }
   ]
)
```

MySQL	MongoDB
`SELECT * FROM empleados WHERE codigo=3`	`db.empleados.find( {codigo:3} );`
`SELECT nombre, email FROM empleados WHERE codigo=3`	`db.empleados.find( {codigo:3}, {nombre:1, email:1} );`
`SELECT * FROM empleados WHERE edad < 40 AND nombre='Rosabel'`	`db.empleados.find( {$and:[ {edad:{$lt:40}}, {nombre:"Rosabel"} ]} );`
`SELECT * FROM empleados WHERE codigo IN (2,4,6)`	`db.empleados.find( {codigo:{$in:[2,4,6]}} );`
`SELECT * FROM empleados WHERE email LIKE '%org'`	`db.empleados.find( {email:{$regex:'.org$'}} );`
`SELECT codigo, nombre FROM empleados WHERE codigo<>5`	`db.empleados.find( {codigo:{$ne:5}}, {codigo:1, nombre:1} );`
`SELECT AVG(edad) AS edadMedia FROM empleados;`	`db.empleados.aggregate( [{ $group: { _id:null, edadMedia: { $avg: "$edad" } } }] );`
`SELECT dep, MAX(edad) AS mayor FROM empleados GROUP BY dep;`	`db.empleados.aggregate( [{ $group: { _id:"$dep", mayor: { $max: "$edad" } } }] );`
`SELECT dep, COUNT(*) AS numEpleados FROM empleados GROUP BY dep;`	`db.empleados.aggregate( [{ $group: { _id:"$dep", numEmpleados: { $sum: 1 } } }] );`

Actualizar registros en una colección

Quizá .update () sea una de las operaciones más completas dentro de MongoDB.

```
db.nombreColeccion.update(
   <criterios>,
   <datos_nuevos>,
   {
     upsert: <boolean>,
     multi: <boolean>
   }
)
```

Veamos un ejemplo sencillo, en el cual indicamos como criterio que la comunidad sea "Murcia" y, a continuación modificamos el valor del campo comunidad por "Región de Murcia".

```
db.comunidades.insert([{comunidad:'Murcia'},{comunidad:'Castilla la Mancha'},
    {comunidad:'Cantabria'},{comunidad:'Comunidad Valenciana'},
    {comunidad:'La Rioja'}])

db.comunidades.find()

db.comunidades.update(
   {comunidad:'Murcia'},
   {comunidad:'Región de Murcia'}
)
```

El parámetro **upsert** a true nos permite indicar que si el criterio no encuentra ningún registro que coincida, los datos nuevos deberán insertarse como un registro nuevo. Por defecto su valor el false.

El parámetro **multi** a true modificará todos los registros que coincidan con el criterio, si es false únicamente modificará el primer registro, generalmente el Id menor. Por defecto su valor el false.

Veamos ahora una serie de ejemplos más complejos para entender la potencia de este comando.

El siguiente comando modificará el registro donde el campo comunidad sea "Comunidad Valenciana", mediante la modificación no vamos a cambiar el dato del campo, sin embargo añadiremos un nuevo campo al documento llamado población con el dato.

```
db.comunidades.update(
   {comunidad:'Comunidad Valenciana'},
   {
     comunidad:'Comunidad Valenciana',
     poblacion: 4953482
   }
)
```

Fíjate que debemos escribir tanto el campo nuevo como los que ya existan, si no hacemos esto se borrarán del registro.

Añadamos ahora el campo provincias en el registro donde el campo **comunidad** sea "Comunidad Valenciana", deberemos añadir un nuevo campo **provincias** con los valores Alicante, Castellón,Valencia y Murcia.

```
db.comunidades.update(
   {comunidad:'Comunidad Valenciana'},
   {
     $set: {provincias: ["Alicante","Castellón","Valencia","Murcia"]}
   }
)
```

A diferencia del ejemplo anterior, hemos utilizado un operador de modificación, en concreto **$set**, este nos permite modificar solo aquellos campos que queramos, por lo tanto no tenemos que reescribir campos que no queramos modificar. Si no existe el campo lo creará.

Bien, como vemos en el comando, nos hemos equivocado al añadir las provincias, hemos añadido una que no pertenece a la Comunidad Valenciana. Para solucionar este fallo, vamos a utilizar otro operador de modificación, esta vez **$pull**, que nos permite eliminar un elemento de una lista simplemente indicando cual.

En este caso eliminaremos un elemento del campo provincias:

```
db.comunidades.update(
   {comunidad:'Comunidad Valenciana'},
   {
     $pull: {provincias: "Murcia"}
   }
)
```

Como podemos ver, basta con indicar el dato que queremos eliminar de la lista, en este caso hemos eliminado "Murcia".

Existen más operadores de modificación, pero no disponemos de tanto tiempo para verlos todos en clase, los puedes consultar aquí.

Operador .update
https://docs.mongodb.com/manual/reference/operator/update/

Eliminar un elemento de la colección

```
db.nombreColeccion.remove({<criterios>})
```

Eliminar una colección

Con la base de datos seleccionada ejecutaremos el siguiente comando para eliminar una colección.

```
db.nombreColeccion.remove({})
```

Si tras este comando sigue apareciendo la colección, puedes ejecutar el siguiente, éste eliminará por completo la colección.

```
db.nombreColeccion.drop()
```

Veamos algunos ejemplos comparando con SQL:

MySQL	MongoDB
`USE tienda;`	`use tienda`

`INSERT INTO`
`  productos(id, descripcion, precio)`
`VALUES`
`  (101,'Manzana royal',1.39);`
	`db.productos.insert({`
	`  "id": 101,`
	`  "descripcion":"Manzana royal",`
	`  "precio":1.39`
	`});`

`INSERT INTO`
`  productos(id, descripcion, precio)`
`VALUES`
`  (102,'Pera blanquilla',1.59),`
`  (103,'Kiwi',3.69);`
	`db.productos.insert(`
	`[`
	`  {`
	`  "id": 102,`
	`  "descripcion":"Pera blanquilla",`
	`  "precio":1.59`
	`  },`
	`  {`
	`  "id": 103,`
	`  "descripcion":"Kiwi",`
	`  "precio":3.69`
	`  }`
	`]`
	`);`

`UPDATE productos`
`  SET precio=3.75`
`  WHERE descripcion='Kiwi';`
	`db.productos.update(`
	`  {"descripcion":"Kiwi"},`
	`  {$set:{"precio":3.75}}`
	`);`

`DELETE FROM productos`
`  WHERE id=102:`
	`db.productos.remove(`
	`  {"id":102}`
	`);`

6.4 Modelado de datos

En este apartado trataremos de "simular" las típicas relaciones que podemos encontrarnos en las bases de datos estructuradas. Como ya se ha comentado, MongoDB es una base de datos NoSQL, no estructurada, por lo que las relaciones son ajenas a ella. Pero podemos modelar los documentos de tal manera que podamos relacionar los datos entre sí.

Podemos encontrar dos patrones de modelado que nos pueden ayudar a relacionar datos que en un modelo estructurado estarían en tablas diferentes.

- **Embebido (Embeber)**: consistirá en incrustar documentos uno dentro de otro, haciéndolo así parte del mismo registro, sería una relación directa. Este patrón sigue el principio de TDA (Tipo de Datos Abstracto).
- **Referenciar**: mediante este método se imitan las claves ajenas para relacionar los datos entre colecciones.

A la hora de aplicar un modelo u otro, deberás pensar muy bien que tratamiento le vas a dar a los datos para ver que método necesitas aplicar. Veamos a continuación las diferentes maneras para modelar las relaciones.

Relaciones de 1 a 1

En este tipo de relaciones, por lo general, se suele reproducir fielmente la normalización para formar una única tabla, aunque encontraremos casos especiales en los que deberemos separar la información.

Vamos a suponer que tenemos una tabla **personas** y una tabla **documentos**, en las que una persona únicamente podrá tener un juego de documentos y un juego de documentos solo puede pertenecer a una persona, por lo tanto, es una relación de 1 a 1.

Si hacemos una traducción directa a MongoDB de esta relación, podemos obtener algo así.

```
Personas = {
  nombre: "Miguel",
  apellidos: "Sánchez",
  telefono: 555345623
}

Documentos = {
  dni: "12341234T",
  seguridad_social: "234298237PRQ"
}
```

Para este tipo de relaciones utilizaremos el método embebido (embeber), obteniendo como resultado lo siguiente.

```
Personas = {
  nombre: "Miguel",
  apellidos: "Sánchez",
  telefono: 555345623
  documentos: {
    dni: "12341234T",
    seguridad_social: "234298237PRQ"
  }
}
```

Relaciones de 1 a N

Veamos ahora un caso en el que tenemos una tabla **persona** y una tabla **vehículos**, de lo que deducimos que una persona puede tener más de un vehículo y un vehículo únicamente puede tener una persona como propietaria.

Veamos en primer lugar la interpretación normal en MongoDB.

```
Personas = {
  nombre: "Miguel",
  apellidos: "Sánchez",
  telefono: 555345623
}

Vehiculo1 = {
  matricula: "1111AAA",
  bastidor: "474uf8e93sie3qx93",
  marca: "Toyota",
  modelo: "Auris",
  puertas: 5
}

Vehiculo2 = {
  matricula: "3344CCF",
  bastidor: "3ifwif823jos9ojj8w",
  marca: "Ford",
  modelo: "Focus",
  puertas: 3
}
```

Para este caso podemos utilizar las dos opciones, veamos como quedarían utilizando ambos métodos.

Utilizando **método embebido** (embedded documents) puede quedar como sigue.

```
Personas = {
  nombre: "Miguel",
  apellidos: "Sánchez",
  telefono: 555345623
  vehiculos : [
    {
      matricula: "1111AAA",
      bastidor: "474uf8e93sie3qx93",
      marca: "Toyota",
      modelo: "Auris",
      puertas: 5
    },
    {
      matricula: "3344CCF",
      bastidor: "3ifwif823jos9ojj8w",
      marca: "Ford",
      modelo: "Focus",
      puertas: 3
    }
   ]
}
```

Veamos ahora **referenciando** (linking), en este caso utilizaremos el _id de cada documento, recuerda que MongoDB puede generar el valor de manera aleatoria, en este ejemplo se utilizan identificadores creados por nosotros.

```
Vehiculo1 = {
  _id: 1,
  matricula: "1111AAA",
  bastidor: "474uf8e93sie3qx93",
  marca: "Toyota",
  modelo: "Auris",
  puertas: 5
}

Vehiculo2 = {
  _id: 2,
  matricula: "3344CCF",
  bastidor: "3ifwif823jos9ojj8w",
  marca: "Ford",
  modelo: "Focus",
  puertas: 3
}

Personas = {
  nombre: "Miguel",
  apellidos: "Sánchez",
  telefono: 555345623,
  vehiculos: [1,2]
}
```

Otra posibilidad sería crear el identificador para la persona y añadirlo en campo como clave ajena en cada documento de vehículo. Esto dependerá del enfoque que quieras o necesites darle a tu base de datos.

Lo más recomendado es utilizar el método embebido, ya que con una simple consulta puedes obtener información más completa.

Relaciones de N a N

Supongamos ahora una relación de este tipo, en la que tenemos las tablas autores y libros, donde nos encontramos con que un autor puede haber escrito uno o mucho libros y un libro puede haber sido escrito por uno o más autores.

Este caso es similar a las relaciones de uno a muchos utilizando el método por referencia, veamos como quedaría.

```
Autor1 = {
  _id: 1,
  nombre: "Màxim",
  apellidos: "Huerta",
  libros: [1000]
}
Autor2 = {
  _id: 2,
  nombre: "Carlos",
  apellidos: "González",
  libros: [1001]
}
Autor3 = {
  _id: 3,
  nombre: "Javier",
  apellidos: "Albusac",
  libros: [1001,1002]
}

Libro1 = {
  _id: 1000,
  titulo: "Mi lugar en el mundo eres tú",
  ISBN: "9788490608517",
  autores: [1]
}
Libro2 = {
  _id: 1001,
  titulo: "Desarrollo de Videojuegos. Un enfoque Práctico.: Volumen 2.
Programación Gráfica: Volumen 2",
  ISBN: "1517413389",
  autores: [2,3]
}
Libro3 = {
  _id: 1002,
  titulo: "Programación Concurrente y Tiempo Real",
  ISBN: "1518608264",
  autores: [3]
}
```

Siguiendo con el ejemplo anterior, si añadimos un campo en la relación, como puede ser el orden de aparición del autor, deberemos utilizar el método embebido, quedando como se muestra a continuación.

```
Autor1 = {
  _id: 1,
  nombre: "Màxim",
  apellidos: "Huerta",
  libros: [
    {
      libro_id: 1000,
      orden: 1
    }]
}

Autor2 = {
  _id: 2,
  nombre: "Carlos",
  apellidos: "González",
  libros: [
    {
      libro_id: 1001,
      orden: 1
    }]
}

Autor3 = {
  _id: 3,
  nombre: "Javier",
  apellidos: "Albusac",
  libros: [
    {
      libro_id: 1001,
      orden: 2
    },{
      libro_id: 1002,
      orden: 1
    }]
}
```

En este caso, hemos embebido en los documentos de autor, y no hemos modificado los documentos de libros. También podríamos haber optado por embeber en libros en vez de en autores.

6.5 GUI para MongoDB

Como ya se ha visto, es bastante sencillo el uso de la consola de MongoDB, pero también podemos encontrar aplicaciones que nos puedan facilitar aún más la gestión de este sistema de bases de datos.

RoboMongo o Robo 3T

RoboMongo está disponible para Windows, OS X y Linux, de fácil instalación y configuración.

RoboMongo
https://robomongo.org/

MongoDB Compass

MongoDB Compass es la GUI para MongoDB. Desde MongoDB 3.2, MongoDB Compass se presenta como la GUI nativa.

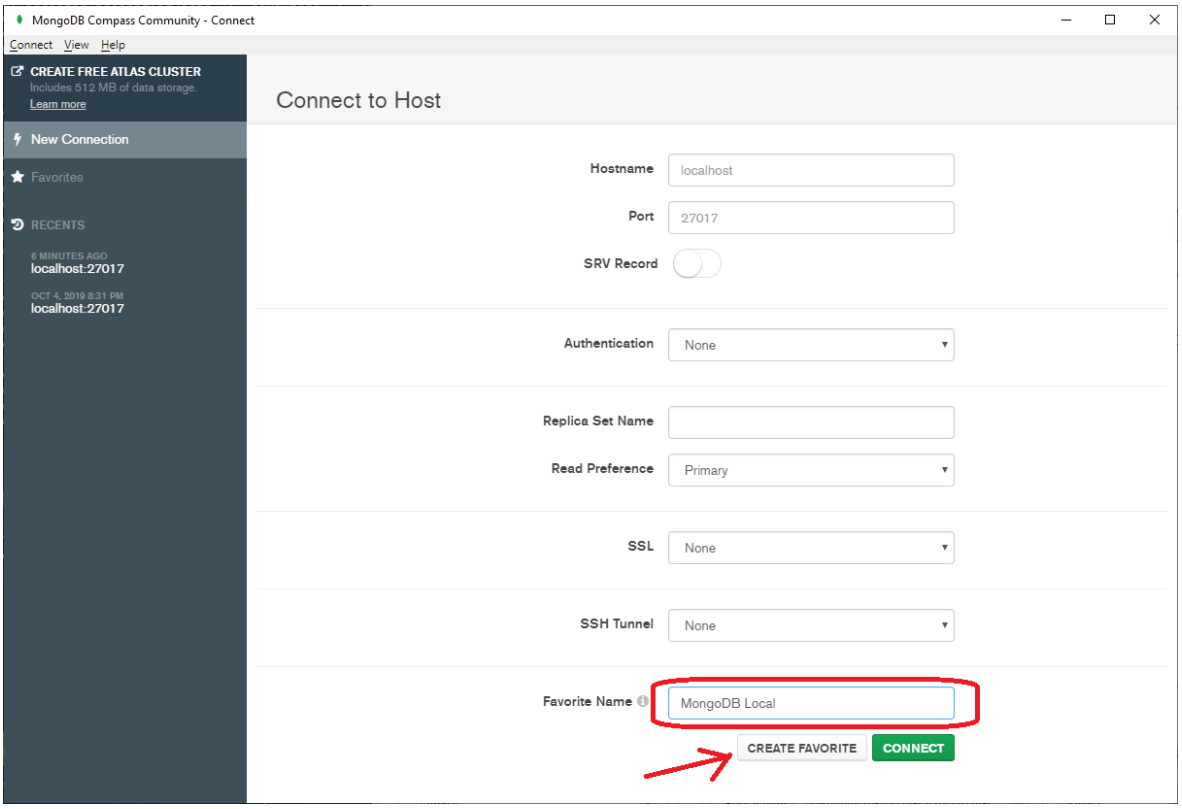
Con MongoDB Compass podrá explorar visualmente sus datos, interactuar sobre la estructura del documento, y realizar consultas, indexación, validación de documentos y más. Las suscripciones comerciales incluyen soporte técnico para MongoDB Compass.

Disponible en Linux, Mac o Windows.

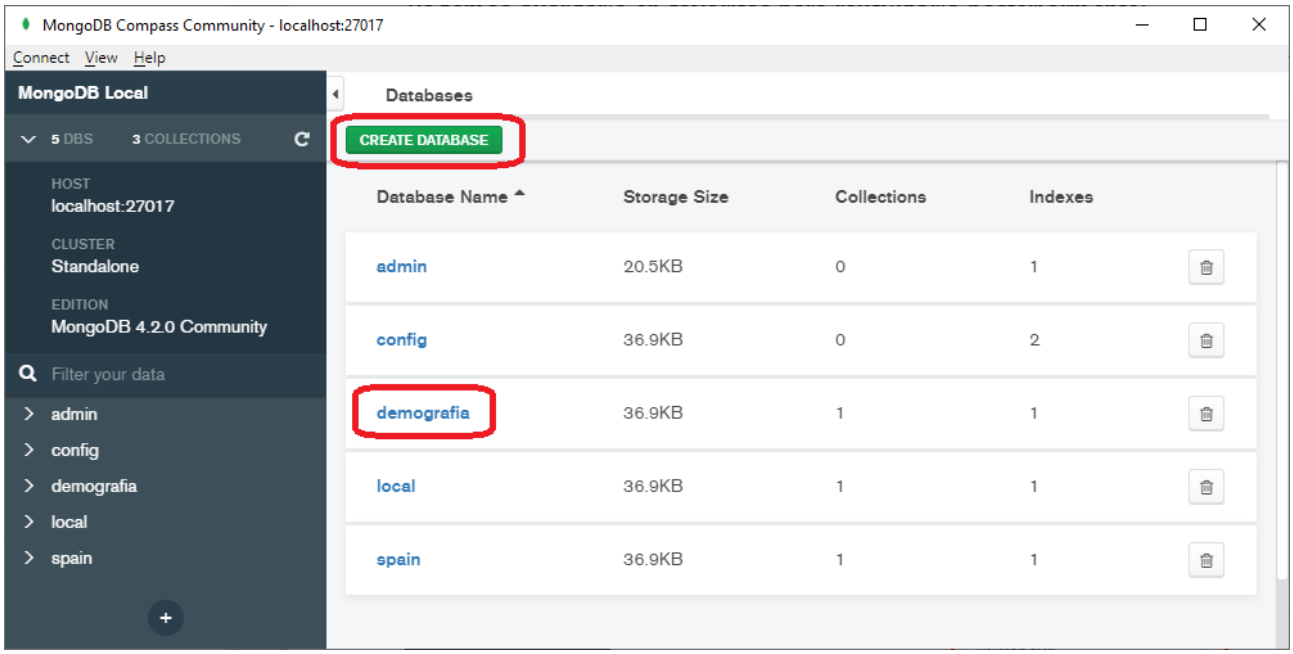
MongoDB Compass
https://www.mongodb.com/products/compass?lang=es-es

Una vez instalado deberemos crear la conexión a nuestro servidor.

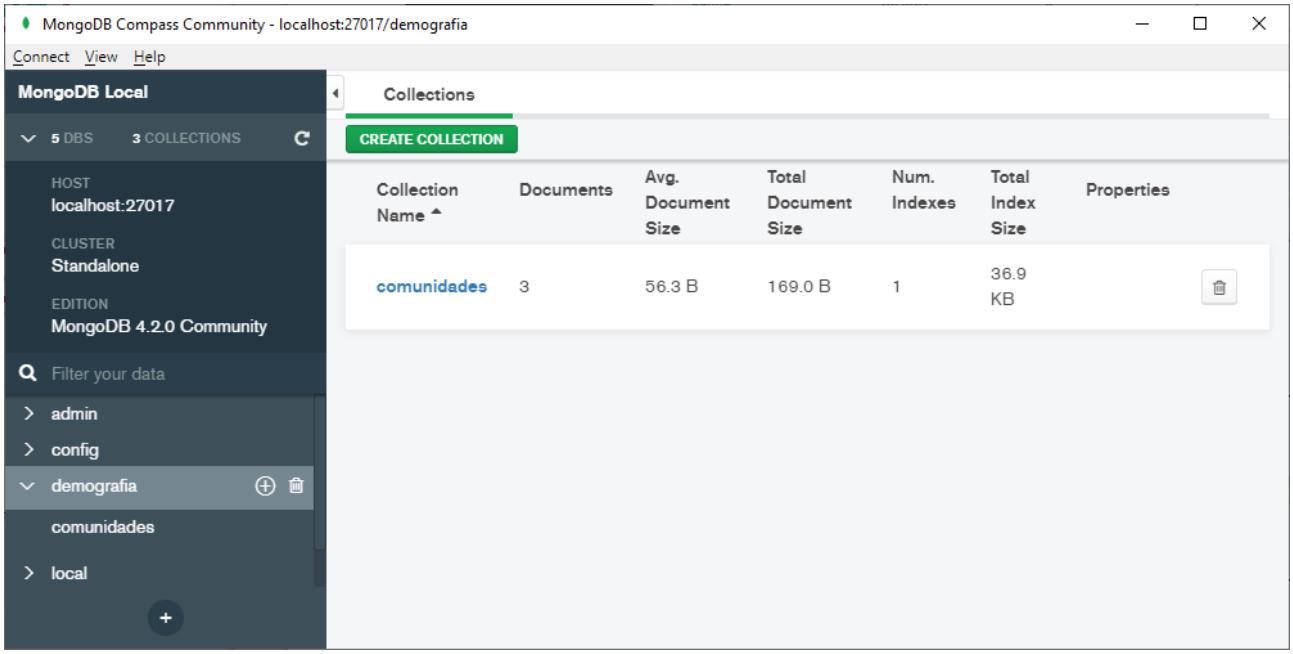
Podemos guardarla en favoritos para reutilizarla posteriormente:



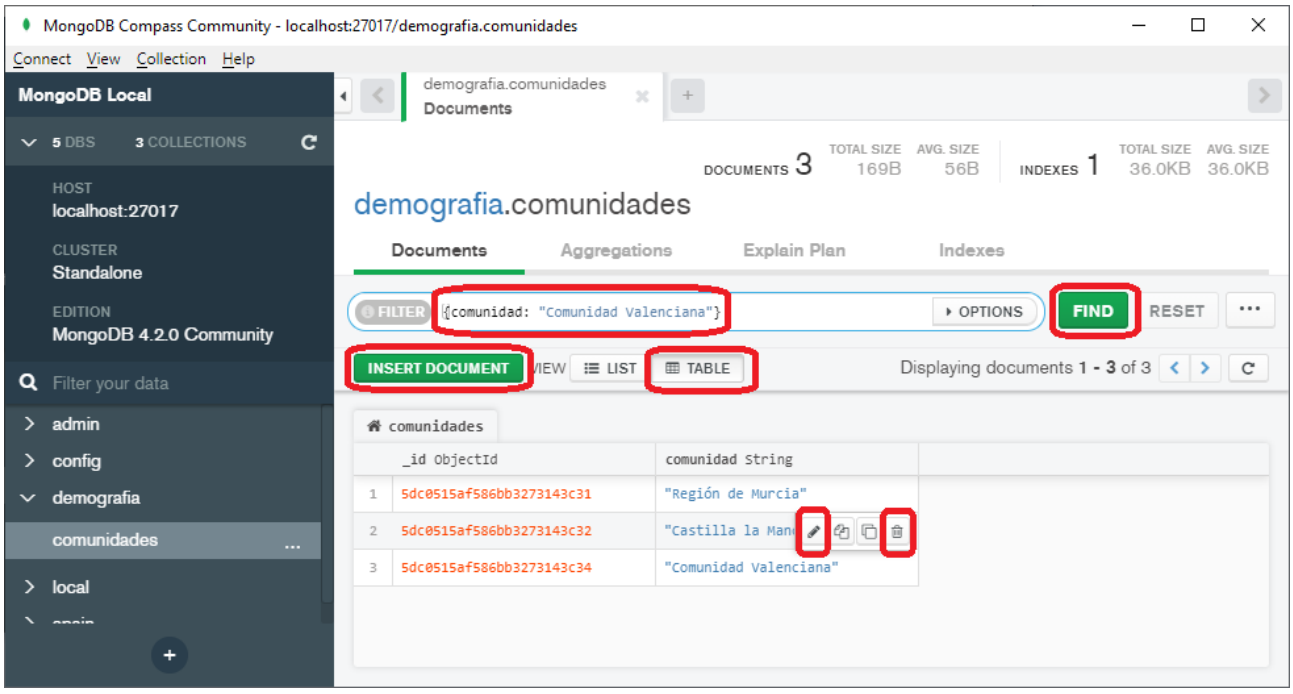
Desde la siguiente pantalla podemos crear una nueva base de datos o seleccionar una ya existente:



Al seleccionar la Base de Datos tendremos las colecciones:



Al pulsar sobre una colección podremos realizar todas las acciones básicas: insertar (insert), buscar (find), editar (edit) y eliminar (delete):



Recomiendo utilizar la vista de tipo "TABLE" pues es más parecida a las usadas en IDE de Bases de Datos Relacionales.

6.6 Scripts para MongoDB

Desde la carpeta de los ejecutables de MongoDB situada en:

```
"C:\Program Files\MongoDB\Server\4.2\bin"
```

Por ejemplo, para mostrar las colecciones disponibles podría evaluar una instrucción de "Mongo Javascript":

```
mongo "mongodb://localhost:27017" --eval "printjson(db.getCollectionNames())"
```

O también preparar un script para para el servidor:

```
mongo "mongodb://localhost:27017/admin" --eval "db.shutdownServer()"
```

Otra opción es ejecutar un conjunto de instrucciones que están escritas en un fichero:

```
mongo "mongodb://localhost:27017" < miscript.js
```

El archivo **miscript.js** contendrá las órdenes de MongoDB a ejecutar. Por ejemplo:

```
// Crear Base de Datos y Colección
use spain

db.comunidades.insert([{comunidad:'Murcia'},{comunidad:'Castilla la Mancha'},
    {comunidad:'Cantabria'},{comunidad:'Comunidad Valenciana'},
    {comunidad:'La Rioja'}])
cls

// Mostrar información
print("*******************************************************")
print("Colecciones en la BD 'spain'")
print("*******************************************************")
db.getCollectionNames()
print()

print("*******************************************************")
print("Datos de la colección 'comunidades' en la BD 'spain'")
print("*******************************************************")
db.comunidades.find().pretty()
print()
```

Con la instrucción **cls** limpiamos la consola y con **print** mostramos un string en la consola. Para más información:

```
mongo --help
```

SCRIPTS para Mongo Shell
https://docs.mongodb.com/manual/tutorial/write-scripts-for-the-mongo-shell/

6.7 Importar/Exportar a CSV o JSON

A continuación vamos a ver dos acciones que nos podrán ayudar con el mantenimiento de nuestro sistema gestor de bases de datos MongoDB, concretamente para realizar backups de nuestras colecciones y restauraciones.

Exportar

El comando **mongoexport** nos permitirá exportar la información de la base de datos y la colección seleccionada a un fichero CSV o JSON.
Veamos su sintaxis y las opciones más comunes.

```
mongoexport <options>

Export data from MongoDB in CSV or JSON format.
--help print usage
-h, --host=<hostname> mongodb host to connect to (setname/host1,host2 for
replica
sets)
-u, --username=<username> username for authentication
-p, --password=<password> password for authentication
-d, --db=<database-name> database to use
-c, --collection=<collection-name> collection to use
-q, --query=<json> query filter, as a JSON string, e.g. '{x:{$gt:1}}'
-o, --out=<filename> output file; if not specified, stdout is used
```

Por ejemplo, si queremos exportar el contenido de la colección **comunidades** que tenemos en la base de datos **demografia** ejecutaremos el siguiente comando.

mongoexport -h localhost:27017 -d demografia -c comunidades -u presidente -o salida.json
(Ejecutar como administrador)

Utilizamos los argumentos:

-h para indicar el servidor y el puerto al que debemos conectarnos,
-d para indicar la base de datos
-c para indicar la colección
-u para indicar el usuario ya que estamos ejecutando el sistema con seguridad
-o para indicar el nombre del fichero JSON que queremos generar

Si no utilizamos -o veremos la salida del comando por la consola.

Tras ejecutar el comando obtendremos un archivo JSON con la información de la colección, si lo queremos como CSV, bastará con cambiar la extensión.

Mongoexport
https://docs.mongodb.com/manual/reference/program/mongoexport/

Importar

El comando mongoimport nos permite realizar la operación inversa, importar información a la base de datos y colección seleccionada desde un fichero CSV o JSON.

Veamos su sintaxis y las opciones más comunes.

```
mongoimport <options> <file>

Import CSV, TSV or JSON data into MongoDB. If no file is provided, mongoimport
reads from stdin.
--help print usage
-h, --host=<hostname> mongodb host to connect to (setname/host1,host2 for
replica
sets)
-u, --username=<username> username for authentication
-p, --password=<password> password for authentication
-d, --db=<database-name> database to use
-c, --collection=<collection-name> collection to use
-f, --fields=<field>[,<field>]* comma separated list of field names, e.g. -f
name,age
--file=<filename> file to import from; if not specified, stdin is used
--drop drop collection before inserting documents
--upsert insert or update objects that already exist
```

El siguiente comando nos permite importar información de un archivo a la base de datos **demografia**, como se puede ver, se creará una nueva colección llamada **comunidades2**.

mongoimport -h localhost:27017 -d demografia -c comunidades2 -u presidente --file salida.json

Si accedemos a la consola de MongoDB podremos ver lo siguiente.

```
> show collections
comunidades
comunidades2
> db.comunidades2.find()
{ "_id" : ObjectId("57a3894233707ff211d26916"), "comunidad" : "Castilla La
Mancha" }
{ "_id" : ObjectId("57a388d233707ff211d26914"), "comunidad" : "Región de
Murcia" }
{ "_id" : ObjectId("57a387e833707ff211d26913"), "comunidad" : "Comunidad
Valenciana" }
> db.comunidades.find()
{ "_id" : ObjectId("57a387e833707ff211d26913"), "comunidad" : "Comunidad
Valenciana" }
{ "_id" : ObjectId("57a388d233707ff211d26914"), "comunidad" : "Región de
Murcia" }
{ "_id" : ObjectId("57a3894233707ff211d26916"), "comunidad" : "Castilla La
Mancha" }
```

Si hacemos la operación sobre una colección ya existente podemos utilizar las opciones –drop o --upsert. La primera eliminará previamente lo que pueda haber en la colección para después insertar los registros nuevos. La segunda es capaz de actualizar los datos de las entradas o registros que coincidan.

Mongoimport
https://docs.mongodb.com/manual/reference/program/mongoimport/

6.8 Almacenamiento de MongoDB

MongoDB utiliza un lenguaje propio al que llamaremos BSON, el cual, podríamos decir que es primo hermano de JSON. Veamos en primer lugar que es JSON para a continuación ver las diferencias y semejanzas entre ambos.

JSON es el acrónimo de JavaScript Object Notation, es un formato de texto ligero y abierto que permite el intercambio de información y, que junto al XML es uno de los principales lenguajes utilizados en la web moderna para el intercambio de información. JSON es además compatible con la mayoría de tipos básicos que podemos necesitar (números, cadenas, valores booleanos, arrays y hashes). Una de las principales ventajas frente a XML es que es más fácil escribir un analizador sintáctico (parser) de JSON.

En MongoDB, este sistema utiliza documentos JSON para almacenar los registros, igual que se utilizan las tablas y filas para almacenar la información en un sistema relacional.

Los resultados arrojados por MongoDB los encontraremos en JSON, por lo que podremos interpretar fácilmente los resultados con una ligera modificación con JavaScript o cualquier lenguaje de programación.

Binary JSON (BSON)

MongoDB utiliza JSON para representar la información, pero la información se encuentra almacenada documentos binarios utilizando BSON, el cual es una extensión de JSON, al cual se le añaden nuevos tipos de datos para poder ser más eficiente en la codificación y decodificación.
En realidad no existen diferencias entre ambos, ya que BSON es una ampliación de JSON. Añade nuevos tipos y está optimizado para un almacenamiento y velocidad más eficiente.

Un ejemplo de documento almacenado en MongoDB con JSON sería el siguiente.

```
{
  '_id' : 1,
  'name' : { 'first' : 'John', 'last' : 'Backus' },
  'contribs' : [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
  'awards' : [
    {
      'award' : 'W.W. McDowell Award',
      'year' : 1967,
      'by' : 'IEEE Computer Society'
    },
    {
      'award' : 'Draper Prize',
      'year' : 1993,
      'by' : 'National Academy of Engineering'
    }
  ]
}
```